ABSTRACT
Following on from My Computer and My Documents, users of SAS® can now also have My Enterprise Guide! The aim of this paper is to present the methods of creating your own add-ins for SAS Enterprise Guide® providing users with rapid exploitation of the power of SAS through a desktop interface.

This paper discusses the advantages of using Enterprise Guide as a platform to deliver to users the ability to achieve your complex business specific tasks. Additionally the paper highlights the full power of the Microsoft Windows environment that is available, in addition to the full power of SAS.

A practical overview is then presented of the steps involved in developing, debugging and deploying Enterprise Guide add-ins with Visual Studio .NET. The demonstration builds an add-in providing customised reporting of subject demographics.

INTRODUCTION
SAS Enterprise Guide is a project based Windows application designed to enable Statisticians, Business Analysts and researchers to rapidly exploit the power of SAS.

To achieve this capability Enterprise Guide presents in excess of 70 point and click tasks that provide SAS functionality, ranging from importing data, through data transformation and visualisation, to sophisticated modelling and forecasting.

So there we are. Our organisation may have enterprise wide, controlled, access to data and readily exploit those data to answer the business choices we are faced with each day. End of story?

Supposing there are highly specific functions performed by a business unit, or statisticians require a highly refined set of analyses needed by your organisation. Whilst SAS 9 is quite capable of performing the calculations and producing the output; Enterprise Guide may not surface that functionality. So how is such functionality achieved when Enterprise Guide is already deployed?

There are two options. The first is Stored Processes, which are able to wrap-up custom code enabling users to supply parameters to the program code with constraints. It is however, difficult to tailor the form presented by a stored process and enforce constraints between parameters on screen. The second option, creating a custom task (or an Add-In) becomes appropriate.

SAS Enterprise Guide supports an extensive object model; that developers of any Microsoft .NET language can harness to create their own add-ins for Enterprise Guide. This paper discusses creating add-ins for SAS Enterprise Guide 3 and newer with Visual Basic .NET and Visual Studio 2003.

GETTING STARTED
To make life easier, SAS Institute has provided a template to the Visual Studio development environment that creates an add-in project to get development started. These templates are available for anyone to download with installation instructions at:

http://support.sas.com/documentation/onlinedoc/guide/customtasks/

The following assumes that the reader has downloaded and installed these template files and is generally familiar with both Visual Studio and Visual Basic.
CREATING A VISUAL STUDIO PROJECT

On creating a new project, the template (highlighted in Figure 1) is available. The SAS template creates a default class (CustomTask1) and a default form CustomTaskForm1. The class contains the signatures for the minimum interfaces which must be implemented. In other words… the developer simply needs to fill in the blanks to define their new Add-In.

![Figure 1: Creating a new project from the SAS Institute supplied template](image1)

![Figure 2: An add-in project ready for development](image2)

After specifying an appropriate name and location, the project is created and the Visual Studio environment now appears as shown in Figure 2. Notice that the appropriate reference has been added making available functionality from the SAS.EG.Addins library.

THE ENTERPRISE GUIDE ADD-IN MODEL

The add-in model provides a series of interfaces that allow the developer to utilise functionality from Enterprise Guide (the host) itself.

An add-in is recognisable to Enterprise Guide because of a class which implements at least the required interfaces for the Enterprise Guide add-in model. In Figure 2, this class is called CustomTask1, in the file CustomTask1.VB. In theory the developer can write any code to perform anything that is possible under the Windows operating system. In practice, most add-ins load and show a Windows form which captures user selections before submitting a defined routine.

WHAT IS AN INTERFACE?

For Enterprise Guide to perform the role of host to an add-in it sets out a series of protocols that must be adhered to. These are the interfaces that the developer must implement.

Enterprise Guide’s add-in model provides several interfaces that can be used to implement various functions. Using interfaces is good practice when unrelated entities (i.e. SAS Institute’s Enterprise Guide code and SAS users’ custom add-ins) need to be assured of working correctly together.

Technically an Interface is a type of object that defines and of methods, properties and events which are implemented by a class.

REQUIRED INTERFACES

A minimum of three interfaces must be implemented for an add-in to be available within Enterprise Guide.

- SAS.EG.AddIns.ISASTask;
  implements methods regarding source data used, SAS code to execute, tables created etc.

- SAS.EG.AddIns.ISASTaskAddIn;
  provides Enterprise Guide with basic name and description information, languages used and an object that is used to reference the add-in host (i.e. Enterprise Guide itself).
• SAS.EG.AddIns.ISASTaskDescription; provides Enterprise Guide with metadata about the add-in such as descriptions, labels and procedures used etc.

These three interfaces and their method stubs are automatically created when the template Enterprise Guide project is used. The developer simply needs to fill in the blanks or edits the default values.

**OBJECT MODEL DOCUMENTATION**

There is a comprehensive object model for building custom add-ins found in the SAS.EG.Addins namespace. The namespace can be navigated by using Visual Studio’s Object Browser window.

At least a familiarity of the available functionality is important, if not practical experience of using many of the methods supplied by the interfaces. Documentation via a CHM file can be located at the following URL given above in the Getting Started section.

**CREATING THE DEMOGRAPHICS ADD-IN**

The class shown to the right is that created by the add-in template, but now setup for a custom task called Demographics.

Additionally two private fields are added which contain long and short descriptions of this task. These are used in the multiple interface methods requiring task names and descriptions.

An object called CodeWriter is instantiated which contains methods used to derive the SAS code ultimately submitted.

The last significant edit to this class is the Show method used to present the add-ins form to the users. The list of possible return codes is extended to trap explicitly the OK, YES or CANCEL button presses.

Recalling that add-ins to Enterprise Guide are simply modal dialogs in V3.0, distinguishing between return codes enables the application to behave appropriately.

These options enable the form to be constructed identically to a SAS supplied add-in, improving consistency for users.

**Hint:** At design time the objects do not appear with the Windows XP themes enabled. Enterprise Guide supports Windows themes quite comprehensively, so ensure that any objects which support the FlatStyle property in the add-in have the value System (rather than the default value of Standard).

Figure 4 (below) shows the My Enterprise Guide solution in Visual Studio .NET with the main form opened. A TreeView object is used to present the categories of functionality to the user. Note that the following properties are set to achieve the view shown: HideSelection = False, ItemHeight=20, Scrollable=False and ShowLines=False. Four root nodes are then added to the node collection, “Task Roles” through to “Results”.

The approach taken presenting the functionality for each category of options is to create one custom control for each of the categories “Task Roles” through to “Results”.

At the point of instantiating the form each of the user controls is added to the main form in the appropriate location, but hidden from view. When a category in the tree is selected the visibility of user controls is toggled appropriately.
A consistent look and feel for column role assignment is achieved by utilising the SASVariableSelector control, which exists in the SAS.EG.Controls library. By adding this control to the Visual Studio toolbox the object can easily be incorporated in a custom add-in, simply by dragging it onto the form. This object is shown at design time below (see Figure 5 below).

It is worth noting that various libraries of SAS functionality are used when a reference is added within the project references area (see Visual Studio Solution Explorer). By default each reference has a property of "Copy Local" with a value of True. This means the functionality already installed by Enterprise Guide is being copied into the \BIN folder of the add-in project.

To prevent this unnecessary duplication the user should set each SAS.EG.*.DLL Copy Local property to False. Any files now created in the \BIN folder of the project will be explicitly created by the add-in project, thereby simplifying deployment.
Preparing the control is performed in two steps:

First, define what roles columns can be assigned to with appropriate attributes.

Second, populate the list of columns to assign as per active data source, i.e. the table selected for use when the task was opened.

Code from each of these stages is considered below.

Here (Figure 6) an instance of the AddRoleParms class is used to add roles to the SASVariableSelector object. Notice that to specify required or optional usage the MinNumVars and MaxNumVars parameters can be controlled. The RoleType property determines what type of column can be assigned to the role.

Once the role is configured it is added to the controls via the AddRole method.

Figure 7 shows partial code used to loop over each column in the active data source. Various properties are determined from the ActiveData property, which is accessed through the consumer object passed into the main form by reference.

In practice the code shown is contained within a Try – Catch block, which elegantly handles any errors which can occur when accessing data sources.

For those familiar with SCL programming, when accessing or reading information from a table, the table must be opened before use and closed after its use is completed.

The methods discussed in Figures 6 and 7 are called as the classes are instantiated.

```vbnet
Dim wrp As New SAS.ES.Controls.SASVariableSelector.AddRoleParms
wrp.Name = "Classification VarRole"
wrp.Type = SAS.ES.Controls.SASVariableSelector.ROLETYPE.All
wrp.MinNumVars = 1
wrp.UseAsRole = True
wrp.LISTERV.DoAddRole(wrp)
wrp.Name = "Number"
wrp.MinNumVars = 0
wrp.UseAsRole = True
wrp.LISTERV.DoAddRole(wrp)
wrp.Type = SAS.ES.Controls.SASVariableSelector.ROLETYPE.All
My.SASVO.AddRole(wrp)
```

Figure 6: Code snippet adding roles to the SAS Variable Selector

```vbnet
With Rs.consumer.ActiveData.Accessor
  .Open()
  For i As Integer = 0 To .ColumnsCount - 1
    Dim wrp As New SAS.ES.Controls.SASVariableSelector.AddVariableFactors
    wrp.Name = .ColumnsInfoByIndex(i).Name
    Select Case i .ColumnsInfoByIndex(i).Group
    Case SAS.ES.Additional.VariableGroup.Character
      wrp.Type = SAS.ES.Controls.VARTYPE.Character
    Case SAS.ES.Additional.VariableGroup.Currency
      wrp.Type = SAS.ES.Controls.VARTYPE.Currency
    Case SAS.ES.Additional.VariableGroup.Numeric
      wrp.Type = SAS.ES.Controls.VARTYPE.Numeric
    Case SAS.ES.Additional.VariableGroup.Geocode
      wrp.Type = SAS.ES.Controls.VARTYPE.Geocode
    Case SAS.ES.Additional.VariableGroup.Data
      wrp.Type = SAS.ES.Controls.VARTYPE.Data
    Case SAS.ES.Additional.VariableGroup.Time
      wrp.Type = SAS.ES.Controls.VARTYPE.Time
    Case SAS.ES.Additional.VariableGroup.ClsData
      wrp.Type = SAS.ES.Controls.VARTYPE.Numeric
    Case Else
      wrp.Type = SAS.ES.Controls.VARTYPE.Numeric
    End Select
    wrp.Label = .ColumnsInfoByIndex(i).Label
    wrp.TextLabel = wrp.Label
    wrp.Format = .ColumnsInfoByIndex(i).Format
    wrp.MaxLength = .ColumnsInfoByIndex(i).Length
    wrp.Informat = .ColumnsInfoByIndex(i).Informat
    wrp.SortBy = .ColumnsInfoByIndex(i).SortPosition
    wrp.SortPosition = 1
    wrp.MyTaskRoles.SAVRO.AddVariable(wrp)
  Next
End With
```

Figure 7: Adding columns from the active data source to the variable selector
The next step is to consider how to create the program that actually gets submitted whenever the project is run, or the user presses “OK”. Enterprise Guide actually requests a string of text from the add-in which is the SAS code to submit. This is done through the SASCode property in the ISASTask interface, hence the developer needs to supply the string passed into the Get method of the property.

In the Demographics add-in there are several steps performed, but it is of a reasonably common structure. A program is written and then parameterised by placing tokens which the add-in replaces with user selected values. The code is stored as an embedded resource within the project which is then streamed into the SASCode property.

A class is written in the project which explicitly performs the role of constructing the string when Enterprise Guide requests it from the SASCode property.

Figure 8 shows the skeleton code which contains tokens of the form %LABEL% which are replaced during parsing.

The WriteCode class takes user selected values and replaces the tokens with actual SAS statements, variable names or other options as appropriate.

The code string is held within an object of type System.Text.StringBuilder, hence the .Replace(“oldString”, “newString”) is used to perform this function.

Figure 8: Embedded SAS code

The final critical step to consider is how values selected by the user are stored for use when the project runs and when the task is edited by the user. The ISASTask interface implements a property called XMLState which defines and retrieves an XML string.

An XML writer object creates XML elements that will be stored in the containing Enterprise Guide project (see the Get method).

Conversely, settings are retrieved through the Set method; an XMLReader object extracts the stored strings and assigns the values in properties defined in the demographics class.

Note that conversion from String to the correct Type may be required here. For example, the code preview ComboBox is a Boolean, which is stored as either “True” or “False” in the XML string.

Figure 9 shows a snippet of the code used in the property method.

An XML writer object creates XML elements that will be stored in the containing Enterprise Guide project (see the Get method).

Conversely, settings are retrieved through the Set method; an XMLReader object extracts the stored strings and assigns the values in properties defined in the demographics class.

Note that conversion from String to the correct Type may be required here. For example, the code preview ComboBox is a Boolean, which is stored as either “True” or “False” in the XML string.

Figure 9: User parameter storage

For example, a property is used to determine the current setting of the Code Preview tick box on the main form. A property (Me.CodePreview) stores this value during the lifetime of the class. The property type is Boolean, i.e. the “Checked” property of the CheckBox object. Hence the XML string is converted into a Boolean with the XMLConvert.ToBoolean method.
DEBUGGING THE ADD-IN
As Enterprise Guide add-ins are simply DLL files, they cannot be started directly. The most effective way the author has found of debugging is to set the start action of the project properties to be the SEGIDE.EXE (i.e. launch Enterprise Guide). Figure 9 shows (for a Visual Basic .NET project) how this is achieved.

On running the Visual Studio project the add-in DLL is compiled before Enterprise Guide is launched.

If the add-in is not already registered within Enterprise Guide it should be done so through the Add-In pull down menu.

The add-in can be run as normal, however any break points set in the Visual Studio project are honoured, enabling debugging to take place as for any other .NET Windows Form application.

Note that Enterprise Guide must be closed before source code can be edited; similarly the project cannot be recompiled if Enterprise Guide is still running as the DLL will be locked.

DEPLOYING THE ADD-INS
When compiled, the Enterprise Guide Add-In will be a library, i.e. DLL file. Deploying an add-in is remarkably simple thanks to the way .NET works. Rather than explicitly needing to register DLL's in the PC's registry a DLL can simply reside in a folder. This means that many versions of the same DLL can co-exist on a single PC without conflict, negating the so called “DLL hell” experienced by those developing COM or ACTIVEX components.

Figure 10: Configuring project properties for debugging

Figure 11 shows the BIN folder within the Visual Studio .NET project and the containing DLL file.

This file needs to be located on the end users PC. Do not attempt to locate the DLL in a shared location for concurrent access as the DLL will appear locked as soon as the first user accesses the add-in.

Figure 12: The deployed demographics add-in

Note that when the add-in is ready for deployment (i.e. your software testing cycle is complete!) the project should be compiled in “Release” mode in Visual Studio.

This optimises the intermediate code held within the DLL and omits the associated PDB file which is used for debugging.

Using the Add-In manager within Enterprise Guide in the normal way will integrate the custom task into the environment. Figure 12 shows the demographics add-in which is included within the Describe category of tasks.

Figure 13 (below) shows the demographics task running within Enterprise Guide.
CONCLUSION

Originally as a statistician and subsequently as a consultant of SAS software I have observed the 80 - 20 rule by people who use SAS. Eighty percent of effort is spent programming and only 20% of effort putting value back into business by the information gained from results of analysis and reporting. SAS Enterprise Guide is a huge step forward towards reversing that ratio. Business users and analysts no longer need to expend large amounts of time and effort training to become competent SAS programmers.

The business specific functionality traditionally delivered to users through SAS/AF® applications is now being easily incorporated into Enterprise Guide through the use of the Add-In model. The model promotes efficient reuse of code thanks to the object-orientated Microsoft .NET languages which, in particular Visual Basic .NET, are easy to learn and develop with thanks to the excellent development environment of Visual Studio .NET.

The author therefore recommends surfacing the power of SAS to users with SAS Enterprise Guide. The ease of use, support for those who may choose to code in the SAS language and the extensibility of the application is provided with true Windows familiarity and flexibility.

The add-in created for this paper can be downloaded from the following URL: http://www.amadeus.co.uk/net

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Author Name:  David Shannon  
Company:  Amadeus Software Limited  
Address:  Orchard Farm, Witney Lane,  
          Leafield, Oxfordshire.  OX29 9PG.  
          ENGLAND  
Phone:  +44 (0)1993 878287  
Fax:  +44 (0)1993 878042  
Email:  david.shannon@amadeus.co.uk  
Web:  www.amadeus.co.uk

Version 1.3  
Copyright (©) 2006 Amadeus Software Limited

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.
Other brand and product names are trademarks of their respective companies.